Europäisches Patentamt | European Patent Office | Office européen des brevets

Bescheinigung | Certificate | Attestation

Die angehefteten Unterlagen stimmen mit der ursprünglich eingereichten Fassung der auf dem nächsten Blatt bezeichneten europäischen Patentanmeldung überein.

The attached documents are exact copies of the European patent application described on the following page, as originally filed.

Les documents fixés à cette attestation sont conformes à la version initialement déposée de la demande de brevet européen spécifiée à la page suivante.

Patentanmeldung Nr. | Patent application No. | Demande de brevet n°

02022692.4

Der Präsident des Europäischen Patentamts;
Im Auftrag

For the President of the European Patent Office

Le Président de l'Office européen des brevets
p.o.

R C van Dijk

Anmeldung Nr:
Application no.: 02022692.4
Demande no:

Anmeldetag:
Date of filing: 10.10.02
Date de dépôt:

Anmelder/Applicant(s)/Demandeur(s):

PACT XPP Technologies AG
Muthmannstrasse 1
80939 München
ALLEMAGNE

Bezeichnung der Erfindung/Title of the invention/Titre de l'invention:
(Falls die Bezeichnung der Erfindung nicht angegeben ist, siehe Beschreibung.
If no title is shown please refer to the description.
Si aucun titre n'est indiqué se referer à la description.)

NML-C-Integration

In Anspruch genommene Priorität(en) / Priority(ies) claimed /Priorité(s) revendiquée(s)
Staat/Tag/Aktenzeichen/State/Date/File no./Pays/Date/Numéro de dépôt:

Internationale Patentklassifikation/International Patent Classification/
Classification internationale des brevets:

G06F9/00

Am Anmeldetag benannte Vertragstaaten/Contracting states designated at date of filing/Etats contractants désignées lors du dépôt:

AT BE BG CH CY CZ DE DK EE ES FI FR GB GR IE IT LI LU MC NL PT SE SK TR

Contents

1 Introduction

NML[2] cores, *i.e.* optimized NML modules/macros to implement a certain operation, function or task, can be embedded in the C source code of a given application. Figure 1 shows the compilation flow when NML modules are used in a C program. Two types of integration schemes are currently supported by the XPP-VC compiler [1]. One refers to the integration of NML modules that release its resources after computing and interface to the C program via array variables (memories). The other approach refers to NML modules, which do not release its resources and interface to the C program via scalar variables.

The name of each NML module must start with "`XPP_`" and there must exist an NML file with the same name where the module is defined. A C header file ("`.h`"), where each module's interface is declared, can be used (other way is to specify the interface declaration in the C program). Internal memories used by the NML module must be pre-placed and the placement information must be declared in the interface declaration. Special pragmas are used to declare the positions of the memories on the XPP [3]. Table 1 shows the pragmas supported and Table 2 shows some pragmas that will be considered in future improvements. The interface specification between the C code and the NML module must be preceeded by the pragma identifying the module or an instance of the module (`#pragma MODULE "<name>"`). Memories used only on the scope of the module must be also declared using `#pragma IRAM <x>, <y>` without a name.

Table 1: Pragmas used to specify the interface between C code and NML modules.

Reference	syntax	Comments	Current Support
NML modules	#pragma MODULE <"name">	define that name refers to an NML module or to an instance of an NML module	yes
specific instances of NML modules	#pragma place <"name"> <X> ,<Y>	manually placement of an instance of an NML module. When the position specified refers to an IRAM the compiler marks such IRAM as used by the current configuration. The name is not neede when the pragma is used after the module invocation.	yes
NML modules	#pragma inline <"name">	instructs the compiler to instantiate the NML module without creating a special configuration for this module	yes (can be used in the first approach)
array variables in internal memories	#pragma IRAM <X>,<Y> <array name>	"X" and "Y" define the IRAM used to accomodate the array	yes
internal memories used by the NML core internally	#pragma IRAM <X>,<Y>	"X" and "Y" define the IRAM used	yes
constants	#pragma CONST <value>	the order of the declaration must be the order of the constants in the NML module.	yes

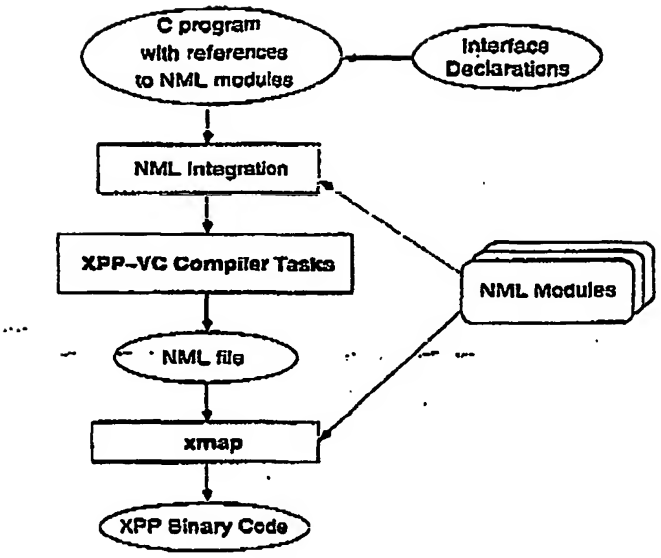

Figure 1: Compilation flow when integrating NML modules in C programs.

Table 2: Pragmas used to specify the interface between NML modules and C code (cont.).

Reference	syntax	Comments	Current Support
array variables in external memories	#pragma EXRAM <Z> <array name> <base address>, <size>	"Z" identifies the I/O port used to interface to the external RAM where the array is located	no (requires XPP-VC support to specify the base address of each array mapped to an external memory)
external memories used by the NML core internally	#pragma EXRAM <Z> <base address>, <size>	"Z" identifies the I/O port used to interface to the external RAM	no (requires XPP-VC support to specify the base address of each array mapped to an external memory)
I/O ports	#pragma IN | OUT | INOUT <Z> <name>	declare an I/O port at position "Z" as input (IN), output (OUT), or input/output (INOUT). Assign a name to that I/O port.	no (should be specified in the interface, but not used by XPP-VC. Used as a form of documentation)

2 Modules not Connected to the NML Structures Generated by XPP-VC

To instantiate a module the user must use the module's name like a normal C function call. The declaration of the module is done as a normal C function plus the pre-defined C pragmas to specify the interface. For each array variable in the function's argument list mapped to IRAMs, a declaration of the location of the IRAM that contains it must be presented using `#pragma IRAM <x>, <y> <array name>`.

In this approach, the compiler only supports as arguments array variables. However, constants (which can be used to communicate parameterizable features: identification of an I/O port, for instance) can be specified using the pragma `CONST`. For internal RAMs, we assume a one-to-one mapping (each internal memory IRAM contains a single array variable). The `place` and `CONST` pragmas can be used after the invocation of the module in the C program. In this case the programmer does not need to use the `MODULE` pragma (see Figure 5).

Each module called in the C code is automatically embedded in the NML output file. By default, the compiler generates one configuration for each call. If instructed by the `inline` pragma NML modules can be embedded in the same configuration in conjunction with structures generated from C code (note that this option can only be used with independent modules, which must be also independent from the C segments of code existent in the same configuration and it is a scheme to include concurrent tasks in the same configuration[1]). Each module used in the C code must self-release its resources after completion of computation. Each module must have only one configuration. Integration of NML applications with more than one configuration into C code must be explicitly done by integrating each of the modules (configurations) individually.

Consider the DCT algorithm shown in Figure 2. It consists of two 8x8 matrix multiplications. Assuming the existence of an optimized NML module to perform the multiplication of two square matrix, the user can re-program the algorithm using the optimized module (see source code in Figure 3). The `XPP_next_conf` in comments in Figure 3 illustrates the configuration boundaries that will be automatically inserted by the compiler to furnish one configuration for each module invocation. The interface declaration for the `XPP_mat_mul` module can be seen in Figure 4. Figure 4(a) shows the definition of N (number of rows or columns of the matrix) which is used to parameterize the NML module (see Figure 4(b)).

In this example, the XPP memory resources are statically pre-defined for each NML module. Thus, to transfer different array variables to distinct instances the user must explicitly copy array elements to the array variables that will be used to communicate data between the program and the NML module (see Figure 3). Thus, all instances in the code of a specific NML module with memories in fixed positions must use the same list of array variables as argument's list.

Another scheme is the use of parameterized memory positions. In this case, for each invocation of the NML module in the C code, the memory positions must be defined according to the initial positions of the internal memories for each array variable (Figure 5 shows a DCT example using a parameterized module to do the matrix multiplication). In this case no movement/relocation of data between internal memories is necessary.

Initialization of array variables in the declarative section of the C program (e.g., int[] a = {1, 4, .., 1};) and used by an NML module is inserted in the first configuration of the application. However, examples

[1]Note that it must be ensured that the same PAE is not used by different independent designs in the same module.

```
// file dct.c
...
// do InIm x CosBlock(T)
for(i=0; i<N;i++) {
   for(j=0;j<N;j++) {
      tmp = 0;
      for(k=0;k<N;k++)
         tmp += InIm[i*N+k] * CosBlock[k+j*N];
      TempBlock[i*N+j] = tmp;
   }
}
// do CosBlock x TempBlock
for(i=0; i<N;i++) {
   for(j=0;j<N;j++) {
      tmp = 0;
      for(k=0;k<N;k++)
         tmp += TempBlock[k*N+j] * CosBlock[i*N+k];
      OutIm[i*N+j]= tmp;
   }
}
...
```

Figure 2: C code for a DCT implementation based on matrix multiplications.

where the NML module is the first configuration in the application require xmap support (this feature is planned).

3 Modules Connected to the NML Structures Generated by XPP-VC

Another possibility is to embed and interconnect NML modules with the NML generated by the XPP-VC compiler. In this case there can exist interconnections between scalar variables of the C code and ports of the NML modules. This is done by using C structs (each struct must have a field with the same name as the related NML module, which must start with "XPP_") to define each NML module and two special functions: `XPP_getmacro` and `XPP_putmacro`. They are used to connect variables of the C code to the input/output ports of the NML module.

Figure 6 shows a segment of code using an NML module to do integer division ("`XPP_div`"). Figure 7 shows the header of the `XPP_div` module and a segment of the NML code generated by the compiler using an instance of that module. Figure 8 shows how to share the same instance of an NML module and Figure 9 shows how to use more than one instance of the same NML module.

Figure 10 presents another example: a C program using XPP FIFOs.

Note that the interface declarations attributed to an NML module instance override previous possible declarations atributed to the name of the NML module.

When it is not possible to synchronize NML module instances with their I/O port connections in a dataflow scheme, an end of completion signal can be used (since it is not possible to declare event signals in C, an integer type must be used):

```
#include "XPPlib.h"
...
const int CosBlock[M] = {...};
const int CosTrans[M] = {...}; // the transpose of CosBlock
...
// XPP_next_conf();
XPP_mat_mult(InIm, CosTrans, TempBlock); //matrix multiplica-
tion in NML
// XPP_next_conf();
// copy the values to the arrays used as arguments of XPP_mat_mult
for(i=0;i<M; i++) {
   InIm[i] = CosBlock[i];
   CosTrans[i] = TempBlock[i];
}
// XPP_next_conf();
XPP_mat_mult(InIm, CosTrans, TempBlock); //matrix multiplica-
tion in NML
// XPP_next_conf();
...
```

Figure 3: A DCT implementation using NML modules to perform the matrix multiplications. Each NML module will use a different configuration.

```
// file XPPlib.h: (a)
...
// The multiplication of two quadratic matrix in NML, function:
void XPP_mat_mult(int A[], int B[], int C[]);
// The specification of the arguments of the NML module
#pragma MODULE "XPP_mat_mult" // identify the module name: us-
ing "<name>"
#pragma IRAM 1,0 A // IRAM <X>,<Y> <array name>
#pragma IRAM 1,1 B
#pragma IRAM 1,2 C
#pragma CONST 8 //number of elements in each row and col-
umn of the square matrix
// other modules:
...
```

```
// file dct.nml: (b)
...
INCLUDE "XPP_mat_mult.nml"
...
MODULE MOD2 {
   OBJ o1: XPP_mat_mult [8] { }
}
...
```

Figure 4: (a) Interface definition for the NML module: XPP_mat_mult ; (b) segment of the NML file generated by the XPP-VC related to the second configuration.

```
#include "XPPlib.h"
...
const int CosBlock[M] = {...};
const int CosTrans[M] = {...}; // the transpose of CosBlock
...
XPP_mat_mult(InIm, CosTrans, TempBlock); //matrix multiplica-
tion in NML
#pragma place 0,0
#pragma CONST 1 //X position of memory with InIm
#pragma CONST 0 //Y position of memory with InIm
#pragma CONST 1 //X position of memory with CosTrans
#pragma CONST 1 //Y position of memory with CosTrans
#pragma CONST 1 //X position of memory with TempBlock
#pragma CONST 2 //Y position of memory with TempBlock
#pragma CONST 8 //number of row and column elements
XPP_mat_mult(CosBlock, TempBlock, OutIm); //matrix multiplica-
tion in NML
#pragma place 0,0
#pragma CONST 1 //X position of memory with CosBlock
#pragma CONST 3 //Y position of memory with CosBlock
#pragma CONST 1 //X position of memory with TempBlock
#pragma CONST 2 //Y position of memory with TempBlock
#pragma CONST 1 //X position of memory with OutIm
#pragma CONST 0 //Y position of memory with OutIm
#pragma CONST 8 //number of row and column elements
...
```

Figure 5: DCT example using NML modules to perform the matrix multiplications. Each NML module will be mapped to a different configuration. In this case an NML module with parameterized memory positions is used.

```
int end_mod;
...
XPP_putmacro(mod1.a, a);
do {
   XPP_getmacro(mod1.end, &end_mod);
} while(end_mod == 0);
...
```

In the example above, a connection is done between the C variable a and the port a of the instance mod1 of an NML module. After that the program wait for the completion of the execution of the instance which is signaled by a value different of zero in the output port end of the NML module instance.

References

[1] João M. P. Cardoso, and M. Weinhardt, "XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture," in *Proc. 12th International Conference on Field Programmable Logic and Applications (FPL'02)*, LNCS, Springer-Verlag, 2002.

```
// original C code (file arraydiv.c): (a)
...
for(i=0;i<M; i++) {
   C[i] = A[i]/B[i];
}
...
```

```
// C code using an NML module to perform the division: (b)
#include "XPPlib.h"
...
divMOD div1;
...
for(i=0;i<M; i++) {
   ai = A[i];
   bi = B[i];
   XPP_putmacro(div1.a, ai);
   XPP_putmacro(div1.b, bi);
   XPP_getmacro(div1.c, &ci);
   C[i] = ci;
}
...
```

```
// file XPPlib.h: (c)
...
// The declaration of the NML module to do integer division
// XPP_div.nml is the name of the NML file where the module
// XPP_div is defined
// computes c = a/b
typedef struct {
   int XPP_div //indentifies the name of the NML module
   int a, b, c;
} divMOD;
#pragma MODULE "XPP_div"
#pragma IRAM 1,0 //it uses an internal RAM in position X=1,Y=0
...
```

Figure 6: Example of embedding an NML module, which performs integer division, into C code: (a) example in C; (b) the same example in stylized C; (c) the description of the NML module in the library.

```
// file XPP_div.nml: (d)
...
// The integer division in NML:
MODULE XPP_div(DIN a, DIN b, DOUT c) {
   // NML code to perform the division
   ...
}
```

```
// file arraydiv.nml: (e)
...
INCLUDE "XPP_div.nml"
...
MODULE ex (
   ...
   // NML code generated by the XPP-
VC to interface to the NML module
   OBJ div1: NML_div ()
   div1.a = <object generated by XPP-VC>.X
   div1.b = <object generated by XPP-VC>.X
   <object generated by XPP-VC>.<A | B> = div1.c
   ...
)
...
```

Figure 7: Example of embedding an NML module, which performs integer division, into C code (cont.): (d) the NML module; (e) the NML module integrated in the NML code of the design.

[2] PACT XPP Technologies, Inc., Germany, "NML Reference: Release 2.0," Technical Report, April. 2001.

[3] PACT XPP Technologies, Inc., Germany, "The XPP White Paper: A Technical Perspective," Technical Report, March, 2002.

```
// original C code (file arraydiv1.c): (a)
...
for(i=0;i<M; i+=2) {
   C[i] = A[i]/B[i];
   C[i+1] = A[i+1]/B[i+1];
}
...
```

```
// C code using a shared NML module to perform both divisions: (b)
#include "XPPlib.h"
...
divMOD div1;
...
for(i=0;i<M; i++) {
   ai = A[i];
   bi = B[i];
   XPP_putmacro(div1.a, ai);
   XPP_putmacro(div1.b, bi);
   XPP_getmacro(div1.c, &ci);
   C[i] = ci;
   ai = A[i+1];
   bi = B[i+1];
   XPP_putmacro(div1.a, ai);
   XPP_putmacro(div1.b, bi);
   XPP_getmacro(div1.c, &ci);
   C[i+1] = ci;
}
...
```

Figure 8: Example embedding more than one NML module instance, which performs integer division, into C code: (a) example in C, which uses two dividers; (b) the example using one NML instance of the divider to perform the two divisions.

```
// C code using two instances of the NML DIVIDER: (c)
#include "XPPlib.h"
...
divMOD div1;
#pragma place "div1" 0,0
#pragma MODULE "div1"
#pragma IRAM 1,1 //the IRAM used when the module is placed in 0, 0
divMOD div2;
#pragma place "div2" 0,6
#pragma MODULE "div2"
#pragma IRAM 1,7 //the IRAM used when the module is placed in 0, 6
...
for(i=0;i<M; i++) {
   ai = A[i];
   bi = B[i];
   XPP_putmacro(div1.a, ai);
   XPP_putmacro(div1.b, bi);
   XPP_getmacro(div1.c, &ci);
   C[i] = ci;
   ai = A[i+1];
   bi = B[i+1];
   XPP_putmacro(div2.a, ai);
   XPP_putmacro(div2.b, bi);
   XPP_getmacro(div2.c, &ci);
   C[i+1] = ci;
}
...
```

Figure 9: Example embedding more than one NML module instance, which performs integer division, into C code (cont.): (c) the example using two NML instances of the divider.

```
// C code using XPP FIFOs (file ex.c): (a)
#include "XPPlib.h"
...
FIFO fifo1;
#pragma place "fifo1" 1,0 //place the fifo in position (1,0)
FIFO fifo2;
#pragma place "fifo2" 1,1 //place the fifo in position (1,1)
...
main() {
   int input_sample_real, delay_value_1,delay_value_2;
   int found_flag = 0;
   int zero_counter = 0;

   while(1) {
     XPP_getstream(1, 0, &input_sample_real);
     XPP_putmacro(fifo1.in, input_sample_real);
     XPP_getmacro(fifo1.out, &delay_value_1);
     XPP_putmacro(fifo2.in, delay_value1);
     XPP_getmacro(fifo2.out, &delay_value_2);

     if((input_sample_real + delay_value_1 +
       delay_value_1 - delay_value_2) == 0) {
        zero_counter++;
        if (zero_counter == 64) {
          found_flag=1;
          zero_counter = 0;
        }
     }
     XPP_putstream(4, 0, found_flag);
   }
}
...
```

```
// file XPPlib.h: (b)
...
// The declaration of the XPP FIFO:
typedef struct {
   int XPP_FIFO // identifies the name of the NML module
   int in, out;
} FIFO;
...
```

Figure 10: Example of a C program using an XPP FIFO; (a) example in C; (b) the description of the NML module in the library.

```
// file XPP_FIFO.nml: (c)
MODULE XPP_FIFO(DIN in, DOUT out) {
   OBJ fifoX: FIFO @ 0,0 {  // relative position used
      IN = in
   }
   out = fifoX.OUT
}
```

```
// file ex.nml: (d)
...
INCLUDE "XPP_FIFO.nml"
...
   OBJ fifo1 : XPP_FIFO @ $1,$0 { }
   fifo1.in = <object generated by XPP-VC>.X
   <object generated by XPP-VC>.<A | B> = fifo1.out
   OBJ fifo2 : XPP_FIFO @ $1,$1 { }
   fifo2.in = <object generated by XPP-VC>.X
   <object generated by XPP-VC>.<A | B> = fifo2.out
...
```

Figure 11: Example of a C program using an XPP FIFO (cont.): (c) the FIFO in NML (relative position is used); (d) the section of the final ex.nml file with the FIFO's instantiations and connections.